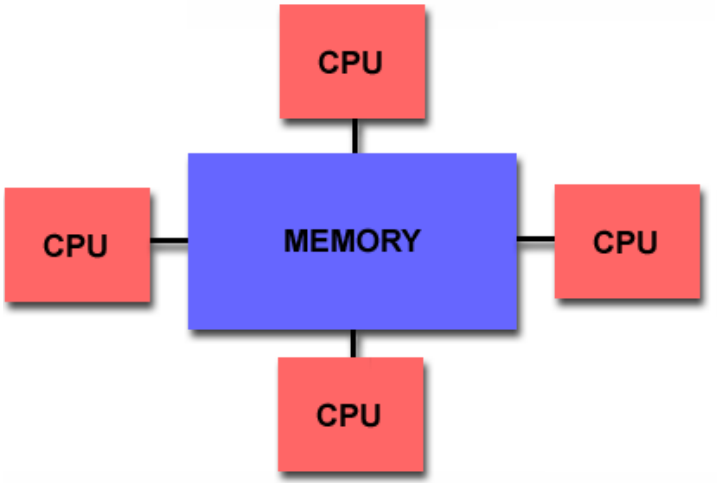
OpenMP

Prof. Michele Amoretti

High Performance Computing 2022/2023

Shared Memory



In parallel computers with shared memory, all processors see the same global address space.

Processors may act independently, but share the same memory resources.

If a processor modifies data in memory, all other processors see it.

The corresponding programming model dictates that all tasks have the same "view" of the memory and are allowed to address the same "logical" locations, independently on where the memory is physically located.

What is OpenMP?

OpenMP adds constructs for **shared-memory threading** to C, C++ and Fortran.

OpenMP consists of **compiler directives** that may include **clauses**, runtime routines and environment variables.

Version 4.0 (July 2013) and 4.5 (November 2015) add support for accelerators (target directives), vectorization (SIMD directives) and thread affinity. Version 5.0 (November 2018) adds extensions to existing directives. Current version is 5.2 (November 2021)

GCC: OpenMP 5.1 is partially supported for C and C++.

INTEL: OpenMP 5.1 C/C++/Fortran is partially supported.

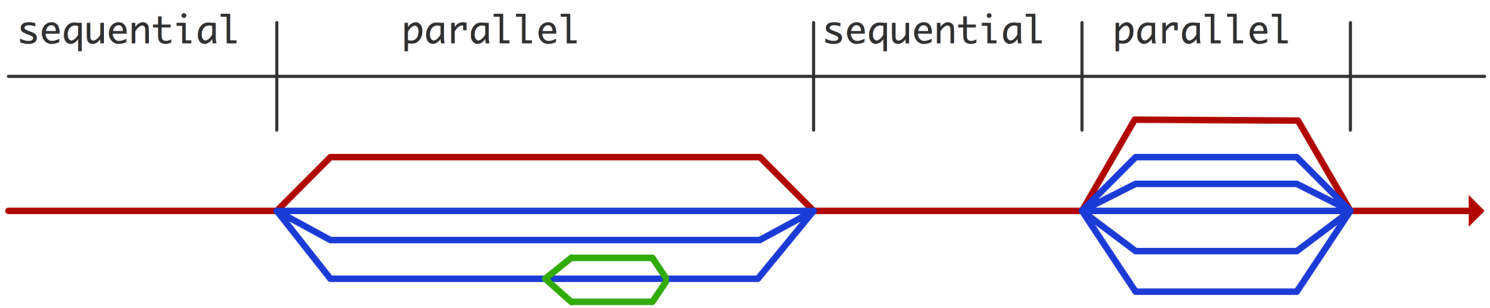
Compile with **-fopenmp** (gcc compiler) or -qopenmp (intel compiler) on Linux.

Execution model

Begin execution as a single process (master thread).

Start of a parallel construct (using special directives): master thread creates team of **threads**.

Fork-join model of parallel execution:



Execution example

ex1.c

```
#include <omp.h>
#include <stdio.h>

int main() {

   //Serial code executed by master thread

   #pragma omp parallel //openMP directive
   {
      // Parallel section executed by all threads
      printf("hello from %d of %d\n", omp_get_thread_num(),
          omp_get_num_threads());
      // omp_get_thread_num() and omp_get_num_threads() are openMP routines
   }

   // Resume serial code executed by master thread
}
```

Run gcc with the -fopenmp flag: gcc -O2 -fopenmp -o ex1 ex1.c

How many threads?

By default, the number of threads coincides with the number of processors of the node.

To control the number of threads used to run an OpenMP program, set the OMP_NUM_THREADS environment variable:

```
% env OMP_NUM_THREADS=3 ./ex1
hello from 2 of 3
hello from 0 of 3
hello from 1 of 3
```

The number of threads can be imposed with an OpenMP routine:

```
omp_set_num_threads(4);
```

OpenMP variables

Variables outside a parallel region are **shared**, and variables inside a parallel region are **private** (allocated in the thread stack).
Programmers can modify this default through the `private()` and `shared()` clauses:

ex2.c

```
#include <omp.h>
#include <stdio.h>

int main() {
    int t, j, i;

    #pragma omp parallel private(t, i) shared(j)
    {
        t = omp_get_thread_num();
        printf("running %d\n", t);
        for (i = 0; i < 1000000; i++)
            j++; /* race! */
        printf("ran %d\n", t);
    }
    printf("%d\n", j);
}
```

It is the programmer's responsibility to ensure that multiple threads properly access shared variables (such as via critical sections).

OpenMP timing

Elapsed wall clock time can be taken using `omp_get_wtime()`.

ex3.c

```
#include <omp.h>
#include <stdio.h>
#include <unistd.h>

int main() {
   double t1,t2;

   printf("Start timer\n");
   t1 = omp_get_wtime();

   // Do something long
   sleep(2);
   t2 = omp_get_wtime();
   printf("%f\n", t2-t1);
}
```

OpenMP directives and clauses

A directive is an OpenMP statement. A directive may include clauses.
Main directives are of 2 types:

- Fork: parallel, for, section, single, master, critical
- Barrier

Syntax:

```
#pragma omp <directive-name> [clause, ..]
{
  // parallelized region
}
//implicit synchronization

#pragma omp barrier //explicit synchronization
```

Clauses are, for example: private, shared, reduction, schedule.

Parallel directive

A **parallel** region is a block of code that will be executed by multiple threads. This is the fundamental OpenMP parallel construct.

Examples:
ex1.c
ex2.c
parallel.c
parallel2.c

For directive

The **for** workshare directive
- requires that the subsequent statement is a for loop;
- makes the loop index private to each thread;
- runs a subset of iterations in each thread.

```
#pragma omp parallel
#pragma omp for
for (i = 0; i < 5; i++)
  printf("hello from %d at %d\n", omp_get_thread_num(), i);
```

Or use `#pragma omp parallel for`

Examples:
for.c
for2.c

Schedule clause

Using just

```
#pragma omp for
```

leaves the decision of **data allocation** up to the compiler

When you want to specify it yourself, use schedule:

```
#pragma omp for schedule(....)
```

Examples:
for.c
for2.c

For directive

for.c

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char* argv[])
{
  int i,j;
  omp_set_num_threads(4);

  #pragma omp parallel  private(j)
  {
    #pragma omp for schedule(static,5)
    for (i=0; i<20; i++)
    {
      j = omp_get_thread_num();
      printf("Execution of thread %d: i=%d\n", j,i);
    }
    printf("%d has finished\n", j);
  }
  return 0;
}
```

Single and Master directives

The **single** directive specifies that the enclosed code is to be executed by only one thread in the team.

The **master** directive specifies a region that is to be executed only by the master thread of the team. All other threads in the team skip this section of code.

Examples:
for2.c
parallel2.c
single.c

```
#pragma omp parallel  private(i,j) shared(k) num_threads(NUMTHR)
  {
    ...
    #pragma omp single // executed by the first free thread
    //#pragma omp master // executed by the master thread
    {
      printf("Execution thread %d enters single mode  \n", j);
      sleep (4);
      printf("Execution thread %d exits single mode  \n", j);
    }
    ...
  }
```

Sections directive

A **sections** workshare directive is followed by a region that has **section** directives, one for each task. There is an implied barrier at the end of a **sections** directive.

```
#pragma omp parallel
#pragma omp sections
{
  #pragma omp section
  printf("Task A: %d\n", omp_get_thread_num());
  #pragma omp section
  printf("Task B: %d\n", omp_get_thread_num());
  #pragma omp section
  printf("Task C: %d\n", omp_get_thread_num());
}
```

Examples:
sections.c
sections2.c

Sections directive

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char* argv[])
{
  int i;
  omp_set_num_threads(3);
  #pragma omp parallel private(i)
  {
    #pragma omp sections
    {
      #pragma omp section
      {
        for(i = 0; i < 5; i++)
        printf("Th %d: %d of section 1\n",omp_get_thread_num(),i);
      }
      #pragma omp section
      {
        for(i = 0; i < 5; i++)
        printf("Th %d: %d of section 2\n",omp_get_thread_num(),i);
      }
      #pragma omp section
      {
        for(i = 0; i < 5; i++)
        printf("Th %d: %d of section 3\n",omp_get_thread_num(),i);
      }
    } // end sections
  } // end parallel
}
```

Safe parallel executions

When can we execute two operations X and Y simultaneously in parallel?

Rules of thumb:

- There must not be any shared data element that is **read** by X and **written** by Y.
- There must not be any shared data element that is **written** by X and **written** by Y.

For example, the following loop **cannot be parallelized**; iteration 0 writes to x[1] and iteration 1 reads from the same element:

```
for (int i = 0; i < 10; ++i) {
    x[i + 1] = f(x[i]);
}
```

Safe parallel executions

The following loop **cannot be parallelized** either; iteration 0 writes to y[0] and iteration 1 writes to the same element:

```
for (int i = 0; i < 10; ++i) {
    y[0] = f(x[i]);
}
```

But parallelizing the following code is perfectly fine, assuming x and y are pointers to distinct array elements and function f does not have any side effects:

```
#pragma omp parallel for
for (int i = 0; i < 10; ++i) {
    y[i] = f(x[i]);
}
```

Critical directive

The **critical** directive specifies a region of code that must be executed by only one thread at a time.

```
#include <omp.h>
main()
{
  int x;
  x = 0;
  #pragma omp parallel shared(x)
  {
    #pragma omp critical
    x = x + 1;
  } /* end of parallel section */
}
```

Reduction clause

The reduction clause of the parallel directive
- makes the specified variable private to each thread
- combines private results on exit

```
int t;
#pragma omp parallel reduction(+:t)
{
  t = omp_get_thread_num() + 1;
  printf("local %d\n", t);
}
printf("reduction %d\n", t);
```

Example:
reduction.c

OpenMP on HPC@UniPR

1) ssh name.surname@login.hpc.unipr.it

2) cp -R /hpc/group/T_2022_HPC_LMI/omp/OpenMP-examples/ .

5) cd OpenMP-examples/

All the examples assume

```
#SBATCH --account=T_2022_HPC_LMI
#SBATCH --partition=bdw
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=8
#SBATCH --time=0-00:5:00
```

OpenMP on HPC@UniPR

For example, assume the source file: **ex1.c**

- Compile with:

  gcc -O2 -fopenmp -o ex1 ex1.c

- Edit omp-run.sh opportunely.

- Run with:

  sbatch omp-run.sh

References

- OpenMP
http://openmp.org/

- OpenMP support in the C/C++ compilers
http://www.openmp.org/resources/openmp-compilers-tools/